MIPP: A Microbenchmark Suite for Performance, Power, and Energy Consumption Characterization of GPU architectures.

Nicola Bombieri
Dept. Computer Science
University of Verona
nicola.bombieri@univr.it

Federico Busato
Dept. Computer Science
University of Verona
federico.busato@univr.it

Franco Fummi
Dept. Computer Science
University of Verona
franco.fummi@univr.it

Michele Scala
Dept. Computer Science
University of Verona
michele.scala@univr.it

***Abstract*—GPU-accelerated applications are becoming increasingly common in high-performance computing as well as in low-power heterogeneous embedded systems. Nevertheless, GPU programming is a challenging task, especially if a GPU application has to be tuned to fully take advantage of the GPU architectural configuration. Even more challenging is the application tuning by considering power and energy consumption, which have emerged as first-order design constraints in addition to performance. Solving bottlenecks of a GPU application such as high thread divergence or poor memory coalescing have a different impact on the overall performance, power and energy consumption. Such an impact also depends on the GPU device on which the application is run. This paper presents a suite of microbenchmarks, which are specialized chunks of GPU code that exercise specific device components (e.g., arithmetic instruction units, shared memory, cache, DRAM, etc.) and that provide the actual characteristics of such components in terms of throughput, power, and energy consumption. The suite aims at enriching standard profiler information and guiding the GPU application tuning on a specific GPU architecture by considering all three design constraints (i.e., power, performance, energy consumption). The paper presents the results obtained by applying the proposed suite to characterize two different GPU devices and to understand how application tuning may impact differently on them.**

I. Introduction

With the growth of computational power and programmability, Graphic Processing Units (GPUs) have become increasingly used as general-purpose accelerators. They not only provide high peak performance, but also excellent energy efficiency [9]. As a consequence, besides supercomputers, GPUs are quickly spreading in low-power and mobile devices like smartphones. NVIDIA Tegra X1 [1] and Qualcomm Snapdragon [2] are some among the several system-on-chip examples available in the mobile market that integrate GPUs with other processing units (i.e., CPUs, FPGAs, DSPs).

On the other hand, the large number of operating hardware resources (e.g., cores and register files) employed in GPUs to support the massive parallelism leads to a significant power consumption. The elevated levels of power consumption have a sensible impact on such many-core device reliability, aging, economic feasibility, performance scaling and deployment into a wide range of application domains. Different techniques have been proposed to manage such high levels of power dissipation and to continue scaling performance and energy. They include approaches based on dynamic voltage/frequency scaling (DVFS) [6], CPU-GPU work division [7], architecture-level/runtime adaptations [14], dynamic resource allocation [5], and application-specific (i.e., programming-level) optimizations [17]. Particularly in this last category, it has been observed that source-code-level transformations and application specific optimizations can significantly affect the GPU resource utilization, performance, and energy efficiency [13].

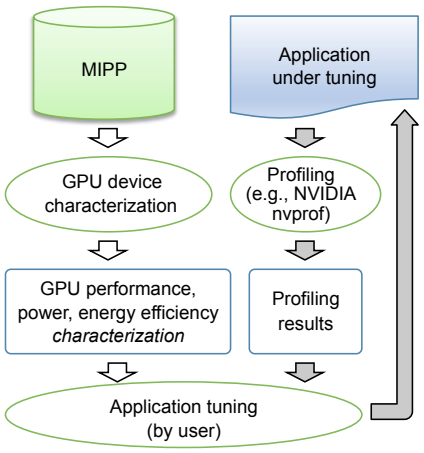


FIG. 1: *Overview of application tuning through MIPP.*

In this context, even tough profiling tools (e.g., CUDA nvprof, AMD APP) exist to help programmers in the application analysis and optimization, they do not provide a complete view of the GPU features (especially on power consumption and energy efficiency) neither they provide a correlation among these design constraints.

For this reason, this paper presents *MIPP*, a suite of microbenchmarks that aims at characterizing a GPU device in terms of performance, power, and energy consumption. In particular, it aims at understanding how application bottlenecks involving selected functional components or underutilization of them can affect the code performance, power consumption, and energy efficiency on the given device. The functional components include arithmetic instruction units, memories (shared, cache, DRAM, constant), scheduling and synchronization components.

Figure 1 shows how the suite can be applied during an

application tuning. First, the microbenchmarks are run on the GPU device to *characterize* the device in terms of performance, power and energy over its main functional components. Then, the application under tuning is profiled by using a standard profiling tool. The profiling results provide information on bottlenecks and underutilization of functional components. The microbenchmark results extend such an information by quantitatively showing how such bottlenecks and underutilization affect performance, power and energy. The proposed model allows the flow (underlined by grey arrows in Figure 1) to be iterated for incremental tuning of the application.

The suite has been applied to characterize two different GPU devices (i.e., an NVIDIA Kepler GTX660 and a low power embedded system NVIDIA Jetson TK1). The results show how the same code optimizations have a different impact on the design constraints on the two different GPU architectures.

The paper is organized as follows. Section II presents the analysis of the related work. Section III introduces the most important concepts and terminology of GPU programming. Section IV presents the microbenchmark suite. Section V reports the experimental results, while Section VI is devoted to concluding remarks.

II. Related Work

Different papers have been presented to evaluate functional and architectural characteristics of GPUs through microbenchmarking.

The first work has been presented in [15], in which the authors developed a microbenchmark suite to measure the CUDA-visible architectural characteristics of the Nvidia GT200 (GTX280) GPU. Such a measure includes various undisclosed characteristics of the processing elements and the memory hierarchies. The analysis exposes undocumented features that impact program performance and correctness. The results can be useful for improving performance optimization, analysis, and modeling on GPU architectures and offer additional insight on the decisions made in developing such a manycore architecture.

In [10], the authors evaluated throughput and the power efficiency of three 128-bit block ciphers (AES, Camellia, and SC2000) on Nvidia Geforce GTX 680 with Kepler architecture and on AMD Radeon HD 7970 with GCN architecture. For the comparison, the authors used Nvidia Geforce GTX 580 and AMD Radeon HD 6770 with architecture of one generation earlier. According to the experimental results, encryption processing on Radeon HD 7970 with GCN architecture designed for general purpose computing engenders extremely high throughput and power efficiency. In contrast, as for GTX 680 with Kepler architecture developed for better power efficiency, arithmetic and logical instructions comprising a large part of block cipher are blushed off. As a result, the throughput of AES-128 on GTX 680 was about 31 % of that on Radeon HD 7970. Power consumption of encryption processing on GTX 680 is certainly lower than that on GTX 580, but the power efficiency was hampered by the low throughput and thereby was no more than 36 % of that on Radeon HD 7970. To investigate the obtained results, the authors applied a microbenchmark suite, which allowed understanding that arithmetic logical instructions are required by encryption processing but are eliminated from some of the processing cores in NVIDIA Kepler architecture, unlike AMD graphics core next (GCN) architectures.

In [16], the authors propose an OpenCL microbenchmark suite for GPUs and CPUs. They present the performance results of hardware and software features such as bus bandwidth, memory architectures, branch architectures and thread hierarchy, etc., evaluated through the proposed microbenchmarks on multi-core X86 CPU and NVIDIA GPUs.

In [4], the authors propose a microbenchmarking methodology based on short elapsed-time events (SETEs) to obtain comprehensive memory microarchitectural details in multi- and many-core processors. This approach requires detailed analysis of potential interfering factors that could affect the intended behavior of such memory systems. They lay out effective guidelines to control and mitigate those interfering factors. Taking the impact of simultaneous multithreading (SMT) into consideration, the methodology not only can measure traditional cache/memory latency and off-chip bandwidth but also can uncover the details of software and hardware prefetching units not attempted in previous studies.

In [8], the authors propose a fine-grained benchmarking approach and apply it on two popular GPUs, namely Fermi and Kepler, to expose the previously unknown characteristics of their memory hierarchies. Specifically, the authors investigate the structures of different cache systems, such as data cache, texture cache, and the translation lookaside buffer (TLB). They also investigate the impact of bank conflict on shared memory access latency. The benchmarking results offer a better understanding on the GPU memory hierarchy, which can help in the software optimization and the modelling of GPU architectures.

Each of these contributions either presents microbenchmarks for characterizing a GPU device from a specific design constraint point of view (performance or power) or presents and analysis of GPU performance and power of a specific application. In contrast, we propose a comprehensive microbenchmark suite that characterizes all the functional and architectural units of a GPU architecture from both the performance and power/energy consumption point of view. Such a characterization allows software developers to understand how application bottlenecks involving selected functional components or underutilization of them can affect the code quality by considering all the three design constraints and the given GPU device.

III. Background

This section summarizes the most important concepts and the corresponding terminology of GPU programming.

GPU devices are massive multithreaded architectures composed by scalable arrays of parallel processors called Streaming Multiprocessors (*SMs*). Each SM contains a set of cores, called Stream Processors (*SPs*). Each SP executes fixed-point and floating-point single-precision operations through dedicated ALU and FPU units. SPs are supported by special purpose units that execute double-precision instructions (*DFU*), transcendental operations (*SFU*), such as trigonometric

functions, and load/store units to issue memory instructions and to calculate memory addresses. The number of SPs per streaming multiprocessor is fixed by the *compute capability* of the device, while the number of DFU, SFU, load/store units depends on the particular chip.

The CUDA thread model consists of four hierarchical levels. A *grid* is composed by *blocks* of threads and each block is divided into groups of 32 parallel thread called *warps* that execute in a SIMD fashion. Threads within a warp that take different execution flows (e.g., due to a control flow instruction) cause *branch divergence* with a consequent instruction serialization and lost of performance. In modern GPU architectures (e.g., NVIDIA Kepler and Maxwell) each SM can handle up to 2048 threads and 64 warps concurrently. The number of warps per SM is called *theoretical occupancy* of the device. Each SM has four warp schedulers, allowing 8 instructions to be execute per clock cycle. Thread blocks are dynamically dispatched to the SMs through a hardware scheduler that works at device-level. The grid configuration and the thread block/warp scheduling strongly affects performance.

Threads of the same block cooperate by sharing data through fast on-chip shared memory. Shared memory is organized in a 32-column matrix (i.e., memory banks). When multiple threads of the same block access different 32-bit words of the same bank, a conflict occurs. Such a *bank conflict* involves re-execution of the memory instructions, with a consequent lost of performance. The GPU memory hierarchy includes also the DRAM, constant, and L2 cache memories, which are visible to all the threads of a grid. The constant cache is a fast and small read-only memory space commonly used to kernel parameter passing and for storing data that will not change during the kernel execution. In contrast, DRAM and L2 cache provide high latency read/write spaces to all threads.

Finally, the access pattern of global memory accesses is critical for the performance. In order to maximize the global memory bandwidth and to reduce the number of bus transactions, multiple memory accesses can be combined into a single transaction. *Memory coalescing* consists of executing memory accesses by different warp threads to an aligned and continuous segment of memory.

IV. The Microbenchmark Suite

We developed a suite of microbenchmarks to selectively study the behaviour of a wide range of GPU functional components. Figure 2 gives an overview of such a microbenchmark suite by reporting, for each microbenchmark, the exercised GPU component, the involved specific instructions, and the considered features.

A microbenchmark consists of a GPU kernel code that exercises a specific functional component of the architecture and whose instructions can be evaluated at a clock-cycle accuracy. The generic structure of the microbenchmark main procedure consists of a long sequence of one or more selected instructions (e.g., arithmetic instructions, memory accesses) that executes without any interference deriving from other instructions. The microbenchmarks have been implemented to stress only a specific functional component at a time, while affecting the others as little as possible to obtain reliable and accurate feedback.

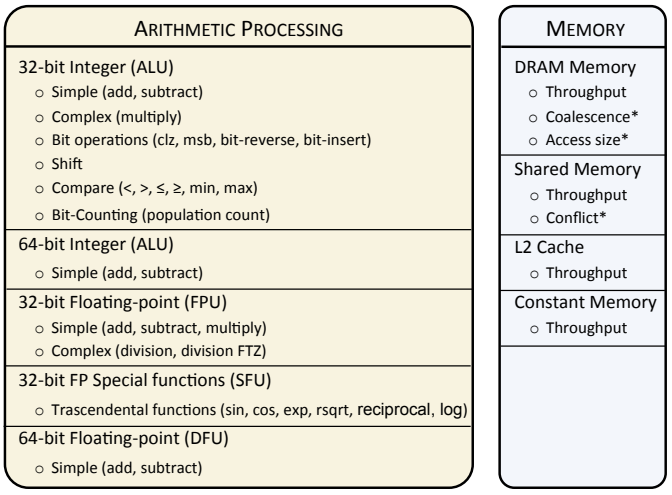


FIG. 2: *Microbenchmark Classes*

The microbenchmark code is written by combining common CUDA C/C++ language with inline intermediate assembly to avoid compiler side effects that may elude the target properties. The parallel thread execution (PTX) is a GPU machine-independent language that allows expressing general purpose computation through virtual ISA. We exploited the PTX language to force a specific operation on a data type, to avoid compiler optimizations, to prevent caching/local-storage mechanisms and to restrict memory access space. Several arrangements have been adopted to preserve the code functionality. As an example, registers are initialized with dynamic values to avoid constant propagation in arithmetic benchmarks. Finally, in order to perform an extensive computation, we apply template meta-programming and nested loops to avoid code-size optimizations and loop collapsing, respectively. The code controls the intensity variability, the amount of computation, and other aspects through parameterized procedures.

Each microbenchmark run returns information like *execution time*, *actual throughput* (to compare with the theoretical throughput from the device specifications), *average* and *max power consumption*, *energy consumption* and *energy efficiency*. Some microbenchmarks (marked with "$*$" in Figure 2) are also applied to exercise functional components with different intensity. As an example, a microbenchmark allows analysing the shared memory throughput by generating a different amount of bank conflicts, from zero to the maximum value, and by measuring the corresponding access time.

Overall, the microbenchmarks provide a quantitative model of the target GPU architecture based on performance and power, and provide important guidelines for the application optimization. As shown in Figure 2, the microbenchmarks are grouped into two classes: *Arithmetic processing* and *memory hierarchy*.

A. Arithmetic processing benchmarks

This class of microbenchmarks targets the complete set of arithmetic instructions natively supported by the GPU, by

```
__global__ ADD_THROUGHPUT()
1: int R1 = clock();   //assign dynamic values to R1,R2 to
2: int R2 = clock();   //avoid constant propagation
3: int startTimer = clock();
4: Computation<N>(R1, R2);   //call the function N times
5: int endTimer = clock();
```

```
template<int N>()   //template metaprogramming
__device__ __forceinline__ COMPUTATION(int R1, int R2)
1: #pragma unroll 4096   //maximum allowed unrolling
2: for (int i = 0; i < 4096; i++) do
3:     asm volatile("add.s32 : "=r"(R1) : "r"(R1), "r"(R2));
4: end   // volatile: prevent ptx compiler optimization
5: Computation<N-1>(R1, R2);   //recursive call
```

FIG. 3: *Example of microbenchmark code. The code aims at measuring the maximum instruction throughput of the* `add` *operation.*

distinguishing between integer and floating point over 32 and 64-bit word sizes.

The benchamrks perform a long sequence of instructions to stress the ALU components. All PTX instructions of arithmetic benchmarks have a direct translation into the native ISA, called SASS (Shared ASSembly), except 64-bit integer operations and floating-point divisions that are compiled into multiple instructions. A SM executes native instructions in one clock cycle, providing a throughput (instructions per clock cycles) limited by the concurrency of the exercised ALU component. Depending on the compute capability of the device and on the architecture, the implementation of non-native instructions may correspond to a different number and type of ISA instructions. Arithmetic benchmarks include also four different types of division operations classified by approximation (IEEE754 Compliance and fast hardware approximation) and normalization (normal and de-normal numbers).

As an example, Figure 3 summarizes the microbenchmark code developed to analyse the 32-bit integer arithmetic processing unit (simple `add`). The code implements dynamic value assignments to registers (see rows 1 and 2 in the upper side of the figure) to avoid the *constant propagation* optimization by the compiler[1]. The code also adopts *recursive* and *template-based* metaprogramming. This allows generating an arbitrarily long sequence of arithmetic instructions without any control flow instructions. ($4096 \times N$ `add` instructions in the example[2]).

B. Memory benchmarks

This class of benchmarks focuses on the impact of throughput and access patterns on DRAM, shared, constant, and L2 cache memories.

The *DRAM throughput* benchmark executes several global accesses to different memory locations with a stride of 128 bytes between grid threads to avoid L1 coalescing. The *L2 benchmark* repeats a compile-time sequence of store instructions on the same memory address. We use cache modifiers

[1]Static value assignments to registers are generally solved and substituted by the compiler optimizations through inlining operations.

[2]In the example, 4,096 unrolls are a good compromise between loop body replication and template recursion. Over a fixed number of loop unrolling iterations the compiler would insert control statements in the loop to reduce the size of the binary code.

```
__device__ clock_t devClocks[RESIDENT_WARPS];
__device__ int devTMP;
template<int CONFLICTS>
__global__ SHAREDMEMCONFLICTS()
1: __shared__ volatile int SMem[1024];
2: volatile int* Offset = SMem + LANE_ID * (CONFLICTS+1);
3: clock_t startTimer = clock64();
4: Computation<N>(Offset);//call the function N times
5: clock_t endTimer = clock64();
6: if (LANE_ID == 0) then
7:     devClocks[WARP_ID] = endTimer - startTimer;
8: if (THREAD_ID == 1024) then
9:     devTMP = SMem[0]; //never executed
```

```
template<int N>   //template metaprogramming
_device_ _forceinline_ COMPUTATION(volatile int* Offset)
1: #pragma unroll 4096
2: for (int i = 0; i < 4096; i++) do
3:     asm volatile("st.volatile.s32 [%0], %1;" : :
4:                  "l"(Offset), "r"(i) : "memory" );
5:          //asm volatile: prevent PTX compiler optimization
6: end
7: Computation<N-1>(Offset);   //recursive call
```

FIG. 4: *Example of the microbenchmark code to measure the impact of shared memory bank conflicts.*

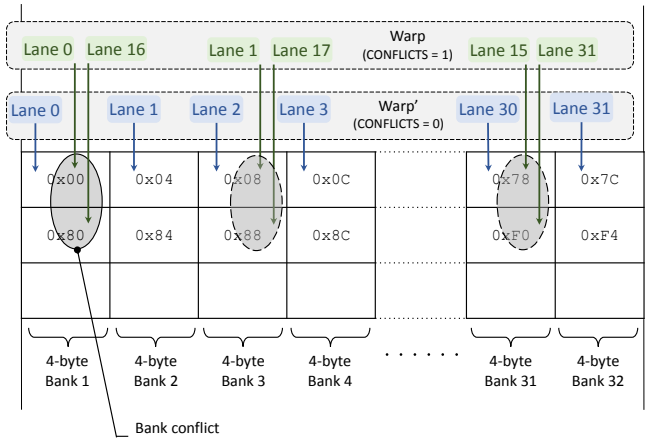


FIG. 5: *Example of memory accesses by threads with no and one bank conflict.*

[11] to avoid L1 cache hits that can occur in the store operations. *Shared* and *constant memory* benchmarks consist of a sequence of load/store instructions respectively. In the *coalescing* benchmark, we vary the number of threads within a warp that access to continuous locations. For example, to test the impact of the worst memory access pattern (no coalescence) we apply the same stride of the DRAM throughput benchmark, to evaluate 1/16 of coalescence we divide the warp threads into 16 groups of two threads where each group accesses in different addresses. The *access size* benchmark copies one large array into another multiple times and, in each execution, we vary the data type size.

Figure 4 summarizes the microbenchmark code developed

to measure the impact of shared memory bank conflicts on the memory access throughput. The procedure first computes, for each thread of a warp, the address offsets of shared memory that can lead to bank conflicts (`line 2` in the upper side of Figure 4, where `LANE_ID` represents the thread id in the warp, and `CONFLICTS` represents the number of conflicts to generate). Figure 5 shows, for example, the offsets generated to lead to no and to one bank conflict. Then, it performs a long sequence of store operations with no interrupt or intermediate operation. The code implements *volatile* quantifiers (`lines 1,2` in the upper side of Figure 4) to avoid *local-storage* optimization by the compiler. As for the simple `add` example, the code adopts *recursive* and *template-based* metaprogramming to generate an arbitrarily long sequence of arithmetic instructions ($4,096 \times N$ `store` instructions in the exampl)e. In the last step (`Line 7` in the upper side of the figure) each warp sends the timing results to the host through global memory. `Line 8,9` ensure that the result is stored in global variable with a fake write instruction (that is never executed since the higher thread id in a block is 1023) to prevent dead code elimination by the compiler.

Similarly, other microbenchmarks of this class exercise their corresponding functional components with different intensity. This allows characterizing the components behaviour under different workloads. For example, the global memory throughput is affected by access pattern that involves a different number of memory transactions or by saturation of the instruction pipeline. Varying the intensity of a microbenchmark allows us to better understand the main factors that affect the performance and the power consumption in real-world applications, where functional components show a wide range of utilization values.

In general, the microbenchmarks have been developed to guarantee enough computation time (i.e., at least of some milliseconds) to overcome the limitation of the sampling frequency in the measurement of the power features ($P_{\max}, \mathrm{P}_{\mathrm{total}}, \mathrm{P}_{\mathrm{avg}}$) and to minimize the RLC effect in the kernel starting/ending phases.

V. Experimental Results

We run the microbenchmark suite to characterize two different GPU devices. The first is an NVIDIA Kepler GeForce GTX 660 with CUDA Toolkit 7.5, AMD Phenom II X6 1055T (3GHz) host processor, and Ubuntu 14.04 operating system. The second is a Tegra K1 SoC (Kepler architecture) on an NVIDIA Jetson TK1 embedded system, with CUDA Toolkit 6.5, 4-Plus-1 Cortex A15 host processor, and Ubuntu 14.04 operating system.

Performance information has been collected through CUDA runtime API to measure the execution time for the memory, scheduling, and synchronization microbenchmarks and through the `clock` device instruction for arithmetic processing microbenchmarks to provide high accuracy.

Power and energy consumption information has been collected through the *Powermon2* power monitoring device [3]. The device allows measuring the voltage and the current values from different sources at the same time with a frequency of 1024 Hz for every sensors. The GTX 660 requires five 12V pins, three for the pci-express power connectors and two for auxiliary connectors. We used a pci-express *interposer* to isolate the GPU power connectors from the motherboard. The Jetson TK1 requires only a DC barrel connector adapter to enable the power monitoring. We designed specific API and procedures to allow microbenchmarks to communicate and to synchronize with the Powermon device. The analysis has been performed with the default GPU frequency setting.

Tables I and II report the results obtained by running the *Arithmetic Processing* benchmarks on the GTX 660 and TK1, respectively. The benchnmarks, which are organized over columns, consist each one of 10^9 instructions per SM (i.e., consider that the GTX 660 consists of 5 SMs, while the TK1 consists of 1 SM). For each benchmark, the tables report the execution time, the theoretical peak throughput of the corresponding functional unit provided in the device specifications [12] ($Spec\ Throughput$) and that measured through the proposed benchmark ($Real\ throughput$). The device specifications do not include the theoretical peak throughput of the Integer 64-bit simple unit since such an operation has not an embedded hardware implementation (it is performed by combining different hardware units).

For both the GTX 660 and TK1, the benchmarks underline that the theoretical peak throughput of several functional units (e.g., complex multiply, population count, shift Integer 32-bit etc.) can be actually reached. The measured peak values often exceed the conservative theoretical values provided by the device specifications. In contrast, the peak throughput of selected functional units, such as the simple 32-bit either Integer or Floating Point `add` cannot be actually fully exploited. We assume this is due to the actual latency of the fetching subunits, which do not support the throughput of the computation subunits. Even though the two devices are fairly different (desktop-oriented GTX 660, and low-power embedded system TK1), the benchmarks underline they rely on equivalent Kepler SMs, whose peak performance are fully comparable.

Finally, Tables I and II report information about power and energy consumption, which are not provided with the device specifications. Power and energy characteristics refer to the whole GPU device and underline the structural characteristics of the two GPU device architectures (5 SMs vs. 1 SM). They also underline that the difference of maximum power among functional units (e.g., `add`, `mult`, `shift`, etc.) is negligible when considering a single SM (see JTK1). It becomes more evident when considering all SMs (e.g., see the difference between FP 32-bit special and Integer 32-bit instructions in GTX 660). The tables also show that FP 64-bit operations have an significant impact on the energy consumption of both the devices (15 times higher than Integer/FP 32-bit simple and Integer compare instructions).

Tables III and IV report the results obtained by running the microbenchmarks of the *memory* class to evaluate the throughput of the different device memories. The results allow understanding how the throughput differs among memories and how it differs between the two devices. As an example, an application running on the TK1 accesses the constant memory 4 times faster than in DRAM. The same application running in the GTX 660 accesses the constant memory 20 times faster than in DRAM. Moreover, the table shows that DRAM accesses strongly affect the average and the max power of GTX

	INTEGER 32-BIT						INTEGER 64-BIT	FP 32-BIT		FP 64-BIT
	SIMPLE	COMPLEX	POP. COUNT	SHIFT	BIT OP.	COMPARE	SIMPLE	SIMPLE	SPECIAL	SIMPLE
Execution Time (ms)	9.6	34.3	34.3	34.3	36.7	9.8	24.5	9.7	36.6	137.0
Spec Throughput (OPs × Cycle) × SM	160	32	32	32	32	160	n.a.	192	32	8
Real Throughput (OPs × Cycle) × SM	126.1	34.7	34.7	34.7	34.3	122.2	51.2	126.0	33.9	8.8
Avg. power (W)	54.4	54.4	53.7	52.5	56.8	54.3	57.6	55.7	60.6	53.3
Max Power (W)	62.0	59.0	56.0	56.0	60.0	59.0	62.0	62	65.0	59.0
Energy (J)	0.5	1.9	1.8	1.8	2.1	0.5	1.4	0.5	2.2	7.3
Energy efficiency (MIPS × Watt) × SM	2,057.1	576.3	583.8	597.0	514.6	2,020.0	760.2	1,990	483.9	146.9
nano Joule per instruction	0.5	1.7	1.7	1.7	1.9	0.5	1.2	0.5	2.1	6.8

TABLE I: *GTX 660 - Characterization with Arithmetic Processing benchmarks.*

	INTEGER 32-BIT						INTEGER 64-BIT	FP 32-BIT		FP 64-BIT
	SIMPLE	COMPLEX	POP. COUNT	SHIFT	BIT OP.	COMPARE	SIMPLE	SIMPLE	SPECIAL	SIMPLE
Execution Time (ms)	122.6	466.2	467.8	466.1	522.4	125.3	368.6	123.9	481.0	1.864
Spec Throughput (OPs × Cycle) × SM	160	32	32	32	32	160	n.a.	192	32	8
Real Throughput (OPs × Cycle) × SM	123.5	32.1	32.1	32.0	29.1	121.9	41.2	123.2	30.7	8
Avg. Power (W)	11.3	11.0	11.0	11.1	11.1	11.3	11.2	11.3	11.3	10.8
Max Power (W)	13.0	13.0	13.0	14.0	13.0	14.0	14.0	13.0	15.0	13.0
Energy (J)	1.4	5.1	5.2	5.2	5.8	1.4	4.1	1.4	5.4	20.2
Energy efficiency (MIPS × Watt) × SM	771.8	208.6	208.1	207.4	184.4	755.3	260.2	765.7	198.0	53.2
nano Joule per instruction	1.3	4.8	4.8	4.8	5.4	1.3	3.8	1.3	5.1	18.8

TABLE II: *Jetson TK1 - Characterization with Arithmetic Processing benchmarks.*

	DRAM	L2	SHARED	CONSTANT
Execution Time (ms)	1,170	1,013	220.2	60.8
Real Throughput (OPs per Cycle)	1.0	1.1	5.3	19.6
Avg. power (W)	92.1	71.5	59.2	63.3
Max power (W)	102.0	80.0	62.0	68.0
Energy (J)	107.8	72.5	13.0	3.8
Energy efficiency (10^6Transactions/Watt)	10.0	14.8	82.4	279.1
nano Joule per transaction	100.4	67.5	12.1	3.6

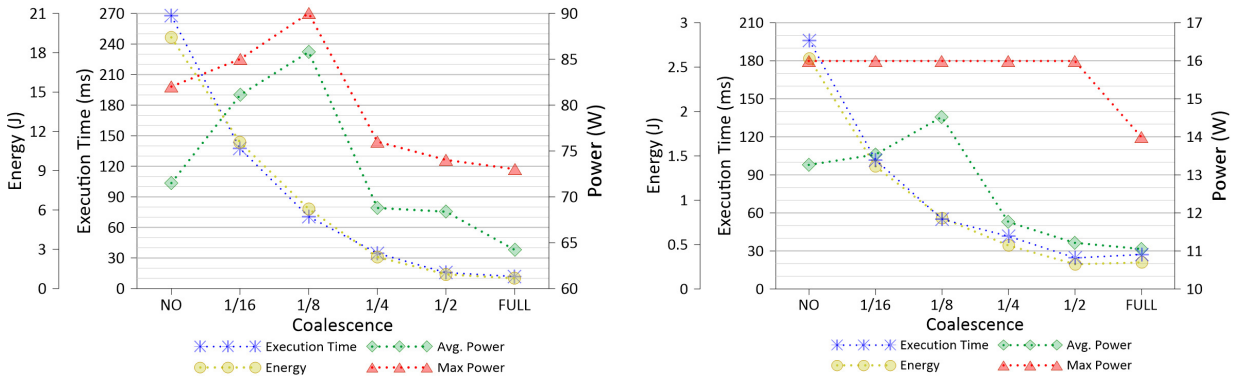
TABLE III: *GTX 660 - Characteristics of accesses on DRAM, L2, shared and constant memories.*

	DRAM	L2	SHARED	CONSTANT
Execution Time (ms)	18,777	16,878	14.915	3,905
Real Throughput (OPs per Cycle)	0.8	0.3	1.0	3.8
Avg. power (W)	14.5	11.5	11.0	11.2
Max power (W)	18.0	15.0	14.0	13.0
Energy (J)	272.1	223.4	164.1	43.9
Energy efficiency (10^6Transactions/Watt)	3.9	4.8	6.5	24.5
nano Joule per transaction	253.4	208.1	152.8	40.9

TABLE IV: *Jetson TK1 - Characteristics of accesses on DRAM, L2, shared and constant memories.*

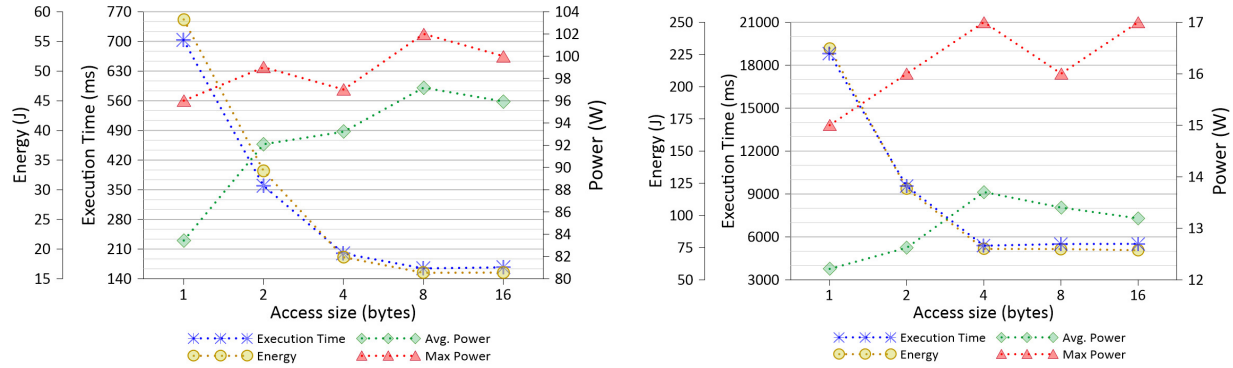
660 and TK1 devices while, the on-chip memories (shared and constant memories) have sightly higher average power than arithmetic instructions.

Figure 6(a) shows the impact of thread *coalescence* in DRAM memory accesses on the GTX 660 performance, power and energy. The figure shows the effect starting from no coalescence (one memory transaction per warp thread access), 1/16 coalescence (one transaction per two warp threads), until FULL coalescence (one transaction per a whole 32-threads warp). The figure shows how performance and energy are proportional to the reached coalescence. In contrast, max and average power reach the highest values at 1/8 coalescence, and they decrease until FULL coalescence. This is due to the



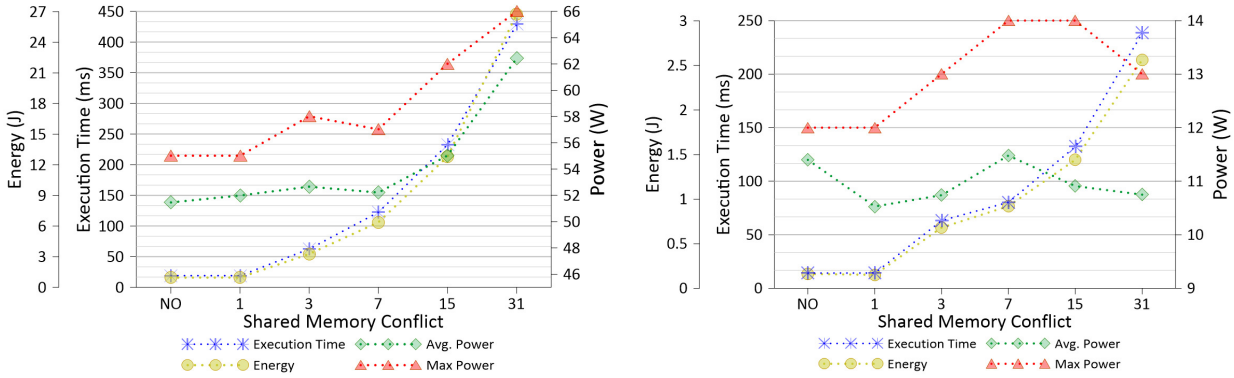
(a) GTX 660 (b) Jetson TK1

FIG. 6: *DRAM coalescence*



(a) GTX 660 (b) Jetson TK1

FIG. 7: *DRAM access size*



(a) GTX 660 (b) Jetson TK1

FIG. 8: *Shared Memory Conflict*

fact that from NO to 1/8, the coalescence is incrementally supported by the 32-Byte L2 memory banks, which saturate at 1/4 coalescence (i.e., each set of 4 transactions per warp, each one 32-Byte large, saturate a 32-Byte L2 bank). From 1/4 on, the coalescence relies on the 128-Byte L1 memory banks. Figure 6(b) reports the same analysis on the TK1, for which the decreasing of the max power can be observed at the FULL coalescence state only.

Figures 7(a) and 7(b) report the impact of size of the thread accesses on DRAM, starting from 1-Byte to 16-Bytes blocks (per thread). The analysis confirms the correlation between performance and energy consumption with average and max power and it quantifies the additional power required to reach the best performance and energy optimizations.

Figures 8(a) and 8(b) quantify the impact of bank conflicts in shared memory on power, performance, and energy consumption. They underline that the bank conflicts similarly impact on performance and energy on the two devices. In contrast the analysis underlines that up to 7 conflicts do not affect the max power on the GTX 660, while up to 7 conflicts strongly affect the max power on the TK1.

Overall, the results obtained by running the proposed suite on a given GPU device allows understanding the specific impact of a tuning step on the design constraints (performance, power, and energy consumption). Improving the code performance that affect a specific functional component (e.g., coalescence of memory accesses) may violate a design constraint on a device, while it may not on a different device (see for instance the different effect on peak power on GTX660 and TK1 by increasing the memory coalescence). Combined with the standard profiler information, the proposed microbenchmark suite can efficiently guide developers in choosing among the possible optimizations during the whole iterative tuning flow.

VI. Conclusions

This paper presented *MIPP*, a suite of microbenchmarks that aims at characterizing a GPU device in terms of performance, power, and energy consumption. *MIPP* aims at understanding how application bottlenecks involving selected functional components or underutilization of them can affect code performance, power consumption, and energy efficiency on a given device. The paper presented the results obtained by applying the microbenchmark suite to characterize two different GPU devices, i.e., an NVIDIA Kepler GTX660 and a low power embedded system NVIDIA Jetson TK1. The results showed how the same code optimizations have a different impact on the design constraints on the two different GPU architectures.

References

[1] NVIDIA Tegra X1. http://www.nvidia.com/object/tegra.html.

[2] Qualcomm Snapdragon. http://www.qualcomm.com/products/snapdragon.

[3] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proceedings of IEEE SoutheastCon (SoutheastCon),*, pages 479–484, 2010.

[4] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. *ACM Transactions on Architecture and Code Optimization*, 11(4):art. n.5, 2015.

[5] S. Hong and H. Kim. An integrated gpu power and performance model. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture*, pages 280–289, 2010.

[6] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *2013 IEEE 31st International Conference on Computer Design, ICCD 2013*, pages 349–356, 2013.

[7] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 48–57, 2012.

[8] X. Mei, K. Zhao, C. Liu, and X. Chu. Benchmarking the memory hierarchy of modern gpus. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8707 LNCS:144–156, 2014.

[9] S. Mittal and J. S. Vetter. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, Aug. 2014.

[10] N. Nishikawa, K. Iwai, H. Tanaka, and T. Kurokawa. Throughput and power efficiency evaluation of block ciphers on kepler and gcn gpus using micro-benchmark analysis. *IEICE Transactions on Information and Systems*, E97-D(6):1506–1515, 2014.

[11] NVIDIA. PTX: Parallel Thread Execution ISA, 2015. http://docs.nvidia.com/cuda/parallel-thread-execution/.

[12] Nvidia CUDA. Programming guide, 2015. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[13] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of ACM SIGPLAN PPoPP*, pages 11–22, 2012.

[14] Y. Wang, S. Roy, and N. Ranganathan. Run-time power-gating in caches of gpus for leakage energy savings. In *Proceedings of ACM/IEEE Design, Automation and Test in Europe, DATE*, pages 300–303, 2012.

[15] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 235–246, 2010.

[16] X. Yan, X. Shi, L. Wang, and H. Yang. An opencl micro-benchmark suite for gpus and cpus. *Journal of Supercomputing*, 69(2):693–713, 2014.

[17] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. Fixing performance bugs: An empirical study of open-source gpgpu programs. In *Proceedings of the International Conference on Parallel Processing*, pages 329–339, 2012.